Exploring Portfolio Scheduling for Long-term Execution of Scientific Workloads in IaaS Clouds

Kefeng Deng, Junqiang Song, Kaijun Ren
School of Computer
National University of Defense Technology
Changsha, China
{dengkefeng,junqiang,renkaijun}@nudt.edu.cn

Alexandru Iosup
Parallel and Distributed Systems Group
Delft University of Technology
Delft, The Netherlands
A.Iosup@tudelft.nl

ABSTRACT

Long-term execution of scientific applications often leads to dynamic workloads and varying application requirements. When the execution uses resources provisioned from IaaS clouds, and thus consumption-related payment, efficient and online scheduling algorithms must be found. Portfolio scheduling, which selects dynamically a suitable policy from a broad portfolio, may provide a solution to this problem. However, selecting online the right policy from possibly tens of alternatives remains challenging. In this work, we introduce an abstract model to explore this selection problem. Based on the model, we present a comprehensive portfolio scheduler that includes tens of provisioning and allocation policies. We propose an algorithm that can enlarge the chance of selecting the best policy in limited time, possibly online. Through trace-based simulation, we evaluate various aspects of our portfolio scheduler, and find performance improvements from 7% to 100% in comparison with the best constituent policies and high improvement for bursty workloads.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed Applications

General Terms

Algorithms, Economics, Management, Performance

Keywords

Portfolio Scheduling, Resource Provisioning, IaaS Cloud, Scientific Workloads

1. INTRODUCTION

The past few years have seen a growing number of scientific computing communities evaluating IaaS clouds for running scientific workloads. For example, several users have tried virtual clusters built with resources leased from commercial clouds such as Amazon EC2 [11, 17]. Other communities have converted their local clusters into public or private clouds [16, 42]. Taking advantages of IaaS cloud resources could hold great promise for scientific computing, especially for highly variable workloads and for parallel jobs which otherwise would have to wait in production queues for long periods of time. To achieve this promise, both scientists and cloud operators still need efficient scheduling algorithms for resource provisioning and job allocation. Although many scheduling algorithms already exist and many have been adapted for IaaS clouds, several studies have shown that none of these algorithms is able to perform well across a wide variety of scientific workload characteristics. To alleviate the problem of selecting a scheduling algorithm, in this work we explore the concept of portfolio scheduling, that is, of selecting a suitable policy from a portfolio, with time limits for the selection process.

Scheduling demanding scientific workloads in the cloud is nontrivial. Genaud et al. [10, 27] study bin-packing strategies for scheduling independent, sequential grid workloads in IaaS clouds. Marshall et al. [25] and Wang et al. [43] present resource provisioning policies for parallel jobs in order to balance monetary cost and job wait time. Other efforts focused on cost-efficient execution of applications such as Bags-of-Tasks (BoTs) [2,28] and scientific workflows [21,22]. To gain deep insight into the performance of scheduling policies, our previous work [41] studies the interplay between provisioning and allocation policies through real experiments. Matching previous studies [13, 14, 35], the results show that no one policy performs the best in all possible situations.

Instead of striving to find better individual scheduling policies, our previous research [8] adapted portfolio scheduling [12] to scientific workloads running in IaaS clouds. The basic idea of portfolio scheduling is to construct a portfolio of scheduling policies, and to select from it the most suitable policy for the current workload and system conditions. The portfolio scheduler evaluates the relative performance of constituent policies though an online, discrete-event simulator. Our previous results have shown that portfolio scheduling can be a viable solution for various workload patterns that include small parallel jobs.

Significantly extending our previous work, in this article we explore a variety of methods not only to reduce the overhead of portfolio scheduling but also to improve the quality of portfolio selection. In comparison with our previous work on portfolio scheduling, we adapt it here for a more demanding and general application model, that is, the long-term execution of small- and medium-scale parallel

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SC13 November 17-21 2013, Denver, CO, USA
Copyright 2013 ACM 978-1-4503-2378-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2503210.2503244

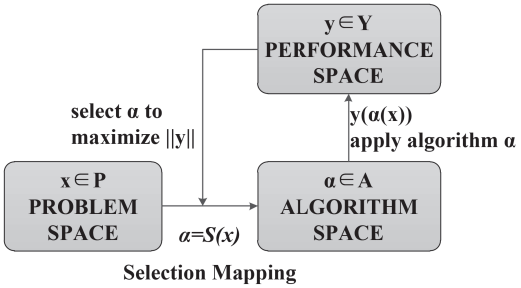


Figure 1: Abstract model for portfolio scheduling, a simplified version of Rice's algorithm selection problem model [29] and the one reproduced by Smith-Miles [34].

workloads observed in real parallel production environments. We address a major limitation to our previous work, which appears in practice when trying to include tens of scheduling policies in the portfolio—evaluating each of them may take too long. Moreover, we explore a variety of parameters that can potentially affect the performance of portfolio scheduling, such as the type of utility function used to select the policy, the time elapsed between selections, or the use of complementary techniques such as job runtime predictions. The main contribution of this work is threefold:

1. We study the use of an abstract algorithm selection model for portfolio scheduling (Section 2). Based on the abstract model, we introduce a portfolio scheduling framework that includes various performance-affecting configuration parameters (Section 3).

2. We propose an algorithm that is able to enlarge the possibility of selecting the best policy from dozens of scheduling policies in the policy portfolio, even when a time constraint is set for policy selection (Section 4).

3. We explore experimentally the concept of portfolio scheduling (Section 6). Through trace-based simulation, we show evidence that our time-constrained portfolio scheduler is able to select a suitable policy with a time budget as small as 200 milliseconds, for a data center that can lease up to 256 concurrent VMs. We also assess the performance impact of various configuration parameters for small- and medium-scale parallel workloads.

2. SCHEDULING MODEL

Portfolio scheduling shares the same abstract model as the general algorithm selection problem [29, 34]. In this abstract model, which is depicted in Figure 1, there are three components that interact in a pre-defined selection process. As shown by the figure, the three components are: the problem space P, which is the set of problem instances needed to be solved; the algorithm space A, which is the set of algorithms used to solve the problem; and the performance space Y, which is the set of performance metrics expected to be optimized.

The process of portfolio scheduling follows a traditional way with four steps: creation, selection, application, and reflection. The creation step constructs that three components in the model. For the problem space P, our work focuses on online scheduling, where the current workload is the only problem instance to be considered. The algorithm space A contains the portfolio of policies used for scheduling. The creation of the portfolio includes a trade-off between the capability to schedule different workload patterns and application types, and the time required to evaluate each policy. In this article, we only consider heuristic-based policies for which the maximum computation complexity is $O(n \log n)$. Their details will be discussed in Section 3.1.

For the performance space Y, we consider in this work various objective functions that can be required by user and operator, expressed as traditional and/or compound metrics. Job slowdown, defined as the ratio between job response time and its runtime, is a widely used performance metric to represent user experience. To eliminate the influence of extremely short jobs on the metric, we use in this work the bounded job slowdown metric [9], which sets a lower bound on the job runtime—following prior work [9], in this work we set 10 seconds as the bound and use the average bounded slowdown BSD as a performance target. We also measure the total runtime of all the jobs (R_J) and the total runtime of all rented VM instances (R_V). Following the cost model of Amazon EC2, VMs are charged by the hour, which means that in our model the runtimes of VM instances are rounded up to the next hour; thus, R_V also denotes the charged cost. The utilization of the scheduler is defined as the ratio between R_J and R_V, and indicates the efficiency of the policies. Resource utilization is an important metric for both data center administrators and users. For system operators, keeping resource utilization high through efficient policies makes them competitive in the market. For users, high utilization means cost efficiency when using the virtual resources.

Although a lower slowdown is to be desired, it may be the result of (much) higher cost. To balance these considerations, we use an extension of a utility function defined in prior work [2, 8, 41]:

$$U = \kappa \cdot \left(\frac{R_J}{R_V}\right)^{\alpha} \cdot \left(\frac{1}{BSD}\right)^{\beta}$$

For this metric, κ is a scaling factor for the total utility, which we set to 100 as in our prior work [8]. The metric parameters α and β are used to express different utility functions: α is used to emphasize the efficiency of resource usage and β is used to stress the urgency of the jobs. By setting different α and β values, our utility function is able to cover the effect of the two main performance requirements in job scheduling field. By setting $\alpha = 0$, the utility function is useful for minimizing job slowdown. By setting $\beta = 0$, the utility function is useful for minimizing the cost or maximizing the utilization of the resources. In this article, we set $\alpha = \beta = 1$ to balance the efficiency and user experience.

The selection step is similar for portfolio scheduling and for the algorithm selection problem. The goal of this step is to find a mapping function S such that the selected scheduling policy $a \in A$ is able to optimize the given performance metric $y \in Y$ for the current workload $x \in P$. Our previous work [8] focused on this problem and solved it by using online simulation as the selection mapping function $S(\cdot)$. Nonetheless, online simulation is a time-consuming function, especially when there are hundreds of jobs in the workload and tens of policies in the portfolio. As a consequence, a major goal of this article is to find methods to

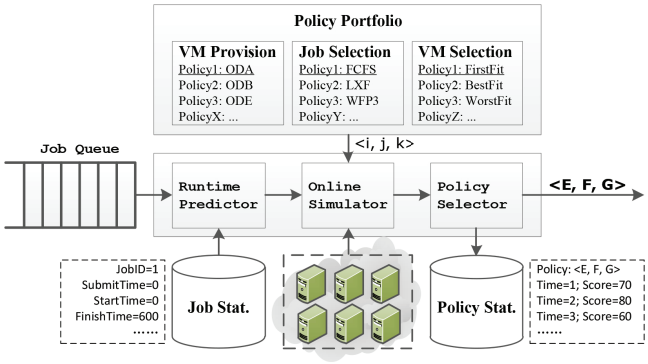


Figure 2: The framework of our portfolio scheduler, a concrete realization of the abstract scheduling model.

reduce the overhead of portfolio simulation while preserving the quality of the selection.

In the application step, the policy that can maximize the utility function in the performance space Y is selected and applied to make real scheduling decisions. The performance of the scheduling policy is collected and stored in the database for reflection. By analyzing the performance of the past selected policies, the reflection step could be useful for improving the quality of selection for future workloads.

We further parameterize the model introduced in this section with an interval between selection decisions and with a time constraint for selecting policies.

3. PORTFOLIO SCHEDULER

Our portfolio scheduler is a concrete realization of the abstract scheduling model discussed in previous section. We design a framework for portfolio scheduling, depicted in Figure 2. The component "Policy Portfolio" implements the algorithm space A, which provides dozens of alternative policies for the online simulator. The "online simulator" uses the provided scheduling policy to run in simulation the jobs currently in the "Job Queue", based on the resource profile of current system; the result of the simulation is a utility score for each evaluated policy. Possibly aided by historical performance data, the "policy selector" chooses the most suitable policy according to the selected reflection criterion. In the remainder of this section, we present the components "Policy Portfolio", the "Runtime Predictor", and the "Online Simulator", in turn.

3.1 The Policy Portfolio

Similarly to our previous work [8, 41], we break the scheduling policies into three parts: the provisioning policies, the job selecting policies and the VM selecting policies. To run the queued jobs, the scheduling policy first provisions a suitable number of VMs from the cloud through the resource provisioning policy, then uses the job selecting policy to order the queued jobs; for each selected job, it selects the required number of appropriate VMs from the cloud according to the VM selecting policy. We populate the portfolio with policies chosen from research work on resource provisioning and scheduling of parallel jobs.

Five provisioning policies are used in our portfolio scheduler. They consider the job wait time (q_i for job i), the job runtime (t_i for job i), and the job parallelism (n_i, the number of processors requested by job i):

1. (The **baseline policy**) *On-Demand All (ODA):* This is a simple, commonly used policy [8, 10, 25, 27, 41]. It leases the required number of VMs for all the queued jobs whenever there are available VMs. This policy is naive: although it may lead to low job slowdown, it also incurs unnecessarily high cost as resources charged for an entire hour may be released after just a few minutes of use.

2. *On-Demand Balance (ODB):* Because scientific workloads may include many short jobs that finish before the hourly charging of resources, it is not necessary to rent new instances for every job. Therefore, the ODB policy tries to keep equal (balanced) the total number of required processors and the number of processors that have already rented. This policy is very similar to the resource management policy presented by the DawningCloud [43].

3. *On-Demand ExecTime (ODE):* The ODE policy is an extension for parallel jobs of the ODE policy we have used in our previous work [8, 41]. It calculates the number of required VMs by rounding the total runtime of all the queued jobs into hours: $\sum n_i \cdot t_i/3600$. The intuition is to pack the jobs as tightly as possible in order to minimize the cost.

4. *On-Demand Maximum (ODM):* This policy leases the maximum number of VMs requested by jobs currently in the queue: $\max(n_1, n_2, \ldots)$. ODM ensures that at least one queued job can be started. The policy is helpful when most of the queued jobs are short, because they can run on already rented VMs instead of leasing new VMs (the ODA policy).

5. *On-Demand XFactor (ODX):* The ODX policy is directly taken from our previous work [8, 41]. It rents the required number of VMs for every job once its bounded slowdown $\frac{q_i+\max(r_i,10)}{\max(r_i,10)}$ exceeds a threshold of 2. This policy can lead to a good trade-off between user experience and resource utilization.

The job selection first orders the job queue based on priority functions, then chooses the first job of the queue for scheduling until insufficient resources remain available. Many job selection policies have been proposed in the literature for various types of jobs. Tan et al. [39] propose several policies that compute job priority (p_i) based on three factors: job wait time, job run time, and job parallelism. Since their proposed policies cover a wide range of policies used in the literature and can avoid job starvation, they are chosen as the candidate job selection policies in our policy portfolio:

1. (The **baseline policy**) *First-Come-First-Serve (FCFS):* FCFS uses the priority : $p_i = q_i$, to order the waiting jobs. This policy is the most commonly used policy for parallel job scheduling and is taken as the baseline policy for job selection.

2. *Largest-Slowdown-First (LXF):* LXF orders the job queue by the slowdown of the jobs: $p_i = (q_i + t_i)/t_i$. In comparison with FCFS, this policy also considers job run time. The intuition is that short jobs suffer more from long wait time than long jobs.

3. *WFP3:* Unlike LXF, which may delay large-scale jobs for a long time, WFP3 takes job parallelism into consideration and prioritizes the job queue by function: $p_i = (q_i/t_i)^3 \cdot n_i$. In addition to preferring large jobs, WFP3 puts more weight on job slowdown since the first factor is cubed.

4. *UNICEF:* In comparison with WFP3, UNICEF goes to the other extreme, by preferring small-scale jobs with short run time. This policy sorts the job queue based on the priority: $p_i = q_i/(log_2(n_i) \cdot t_i)$. It tries to offer quick response time for small jobs.

Although we lease one type of VMs for all the jobs, there is still a need to choose a proper set of VMs for each selected job. The reason is that idle VMs may have different remaining time until they get charged for the next hour. Based on different consideration of the remaining time, we use for VM selection three policies that are originally used to solve the online bin-packing problem [6, 10]:

1. (The **baseline policy**) *First Fit (FF):* FF is the simplest VM selection policy in our study. FF chooses idle VMs without distinction. The advantage of this policy is speed, especially in comparison to policies that sort the VMs.

2. *Best Fit (BF):* For each selected job, BF chooses the VMs such that the remaining time of the selected VMs after running the job is minimal across available VMs. This policy aims at reducing the charged cost by improving the utilization of the VMs.

3. *Worst Fit (WF):* Conversely, WF selects the VMs that will have the maximum remaining time after running the job. The goal of this policy is to balance the usage of each VM. By leaving as much idle time as possible, a large job arriving in the future may be more easily allocated to existing VMs.

Overall, we get a total of 60 scheduling policies in our policy portfolio. Given m jobs and n VMs, the maximum computation complexity of VM provisioning is $\mathcal{O}(m)$ and the maximum computation complexity for job allocation is $\mathcal{O}(m \log m) \cdot \mathcal{O}(n \log n)$. Thus, the maximum computation complexity of the scheduling policies is $\mathcal{O}(mn \log mn)$.

3.2 The Runtime Predictor

Job runtime is an indispensable information in our portfolio scheduler, not only because it is used in some of the resource provisioning and job selection policies, but also for the simulation of policies by the online simulator (see Section 3.3). Though users are required to provide estimated runtime for their jobs in many computer systems, their estimates are known to be highly inaccurate [40, 44]. Therefore, Tsafrir et al. [40] suggest replacing user estimates with system-generated predictions for parallel job scheduling. In particular, they use the average runtime of the two most recently submitted and completed jobs from the same user as the predicted runtime for the new job.

In our portfolio scheduler, we use the algorithm suggested by Tsafrir et al. [40] to adjust user estimates. The algorithm is an instance of the widely used k-nearest neighbor (k-nn) algorithm [33]. Experimental results have shown k=2 is the optimal operation window for the evaluated workloads with an accuracy around 50% [36, 40]. A thorough study of job runtime prediction is beyond the scope of this work; we refer to [26] for more sophisticated algorithms. Nevertheless, the experimental results in Section 6.3 show that even for such an inaccurate predictor, our portfolio scheduler can still perform well.

3.3 The Online Simulator

The online simulator represents the selection mapping between the scheduling policy and the performance target in the abstract scheduling model introduced in Section 2. We implement the simulator as a function of the queued workload (the queued jobs), the cloud profile (the information of running and idle VMs in the current system), and the scheduling policy. The simulator uses the scheduling policy for resource provisioning and allocation until all the jobs in the workload are finished. After that, it outputs the utility score of the scheduling policy and the simulation cost. This process continues until all the scheduling policies are evaluated or a time condition interrupts the process.

As mentioned previously, online simulation is an expensive mapping function. For tens of scheduling policies in a single portfolio, the time taken to evaluate all of them could be prohibitive—when the number of queued jobs and of candidate VMs is in the order of hundreds to thousands, the portfolio scheduler cannot take sub-second decisions. To solve this problem, we introduce a time constraint and present a time-constrained portfolio simulation algorithm to limit the execution of the online simulator in the following section.

4. TIME-CONSTRAINED SIMULATION

To make sure the scheduling decisions are made in time, we set a constraint Δ to limit the time spent for any single simulation of the entire portfolio. To maximize the chance of selecting the best policy for the current workload, we categorize the scheduling policies into three sets: *Smart*, *Stale* and *Poor*. The policies in the *Smart* set have obtained top utility-function scores in the previous portfolio simulation. The policies in the *Pool* set have performed the worst in the former simulation. The *Stale* set contains policies from the *Smart* set and the *Poor* set that have not been simulated in the previous simulation.

We design an algorithm for portfolio scheduling with time constraints that uses the three sets of policies *Smart*, *Stale*, and *Poor*. Algorithm 1 summarizes the pseudo-code of the algorithm, which includes three phases. The first phase allocates time quotas for the three sets. As shown by Algorithm 1 line 1-2, Δ is first partitioned proportionally to the number of policies in *Smart*, *Stale* and *Poor*. When the algorithm is first invoked, all the policies are put into *Smart* set. Hence, both *Stale* and *Poor* are empty, and *Smart* gets the entire Δ time for simulation.

The second phase simulates the sets of policies under the limitation of their time quotas (Algorithm 1 line 3-19). The policies in *Smart*, *Stale*, and *Poor* are simulated, in this order. The procedure *simulate* receives as input the queue of batch jobs (*Queue*), the resource state of current cloud ($Profile$), and the scheduling policy (P_i); *simulate* returns the utility score of the scheduling policy (s_i) along with the simulation time (c_i). For *Smart* and *Stale*, the policies are simulated sequentially until their quotas run out. For *Poor*, the algorithm randomly selects a policy, each time. All the

Algorithm 1 Time-Constrained Portfolio Simulation

Input: $Queue$, the set of batch jobs; $Profile$, the state of current system; $Smart$, the set of smart policies; $Stale$, the set of stale policies; $Poor$, the set of poor policies; λ, the selection ratio

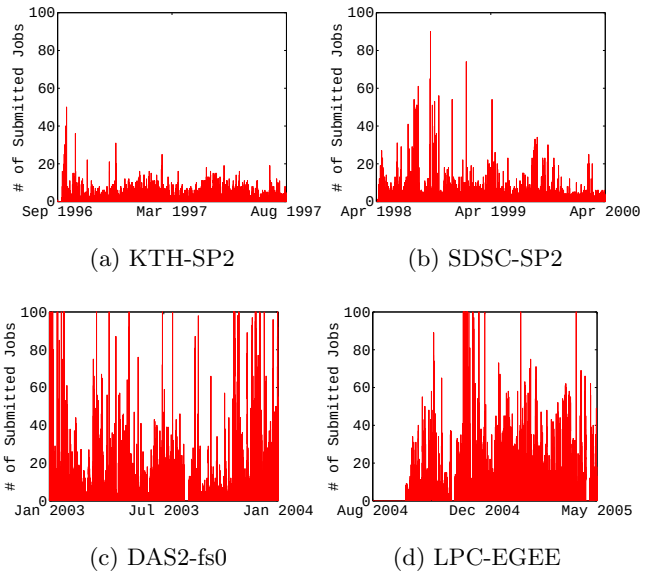
Output: The *best* policy for current workload

1: $Q \leftarrow \varnothing$, $N = \|Smart\| + \|Stale\| + \|Poor\|$;
2: $\delta_1 = \frac{\|Smart\|}{N} \cdot \Delta$, $\delta_2 = \frac{\|Stale\|}{N} \cdot \Delta$, $\delta_3 = \Delta - (\delta_1 + \delta_2)$;
3: **for** $P_i \in Smart$, $i = 1, 2, \cdots, \|Smart\|$ && $\delta_1 > 0$ **do**
4: $s_i, c_i = simulate(Queue, Profile, P_i)$;
5: $Smart \leftarrow Smart - \{P_i\}$, $Q \leftarrow Q \cup \{P_i\}$;
6: $\delta_1 = \delta_1 - c_i$;
7: **end for**
8: **for** $P_i \in Stale$, $i = 1, 2, \cdots, \|Stale\|$ && $\delta_2 > 0$ **do**
9: $s_i, c_i = simulate(Queue, Profile, P_i)$;
10: $Stale \leftarrow Stale - \{P_i\}$, $Q \leftarrow Q \cup \{P_i\}$;
11: $\delta_2 = \delta_2 - c_i$;
12: **end for**
13: $\delta_3 = \delta_3 + \delta_2 + \delta_1$;
14: **while** $\delta_3 > 0$ **do**
15: $i = Random.nextInt(\|Poor\|)$;
16: $s_i, c_i = simulate(Queue, Profile, P_i)$;
17: $Poor \leftarrow Poor - \{P_i\}$, $Q \leftarrow Q \cup \{P_i\}$;
18: $\delta_3 = \delta_3 - c_i$;
19: **end while**
20: $Stale \leftarrow Stale \cup Smart$, $Smart \leftarrow \varnothing$;
21: $Selector.sort(Q)$;
22: $Smart \leftarrow \{P_i \in Q | i = 1, 2, \cdots, \lambda\|Q\|\}$;
23: $Poor \leftarrow Poor \cup \{Q - Smart\}$;
24: **return** $Smart.first()$;

simulated policies are moved out from their original sets and put into a temporary set Q.

The final phase is to rearrange the policies and return the best policy for scheduling the current workload (Algorithm 1 line 20-24). Firstly, the remaining policies in $Smart$ are put into the end of $Stale$ such that the policies in $Stale$ are simulated in accordance to their staleness. After that, the algorithm calls $Selector$ procedure, which sorts the simulated policies based on their utility score and puts the top $\lambda\|Q\|$ policies into $Smart$. The remaining $(1-\lambda)\|Q\|$ policies, which had exhibited lower performance in the online simulator, are added into the $Poor$ set. The score of the simulated policies is also stored in the database for reflection, which we will explore in the future work. Since the first policy in $Smart$ has the highest performance, it is selected as the best policy and returned to portfolio scheduler for real scheduling.

Our algorithm has an *stabilization* property: the number of policies in each set will approximately stabilize at $\|Smart\| = \lambda K$, $\|Stale\| = \lambda(N-K)$, and $\|Poor\| = (1-\lambda)N$. We now provide an informal proof. Assume that an average of K policies can be simulated in the given time constraint Δ. Thus, for a single policy, it takes $\mu = \frac{\Delta}{K}$ time to be simulated. After the first invocation of Algorithm 1, there will be λK, $(1-\lambda)K$ and $(N-K)$ policies in $Smart$, $Poor$ and $Stale$ respectively. In the second invocation, $Quota = \frac{\lambda K}{N}\Delta$ time is allocated to $Smart$, and $\frac{Quota}{\mu}$ policies will be simulated. This means $\lambda K(1-\frac{K}{N})$ policies will be removed from $Smart$ and put into $Stale$. Suppose at a particular time, $Stale$ contains x policies.



(a) KTH-SP2 (b) SDSC-SP2

(c) DAS2-fs0 (d) LPC-EGEE

Figure 3: The number of submitted jobs during ten-minute intervals in four small-to-medium parallel computer systems. All traces show distinct workload patterns. The vertical axis is limited to 100 for better visibility.

Table 1: The characteristics of the workload traces, including the total number of jobs, the number of jobs that require no more than 64 processors (count and percentage).

Trace#. Name	Time [mo.]	Jobs			CPUs	Load [%]
		–	≤ 64	%		
T1. KTH SP2	11	28,480	28,158	98.9	100	70.4
T2. SDSC SP2	24	53,911	53,548	99.3	128	83.5
T3. DAS2 fs0	12	215,638	206,925	96.0	144	14.9
T4. LPC-EGEE	9	214,322	214,322	100	140	20.8

Then, $Quota = \frac{\Delta}{N}x$ time is allocated to $Stale$ by which $\frac{K}{N}x$ policies will be simulated and removed. Among the simulated policies, the well-performing policies will be put into $Smart$ and the poorly performing policies will be placed into $Poor$. This process will continue until $\frac{K}{N}x$ is equivalent to $\lambda K(1-\frac{K}{N})$, or $x = \lambda(N-K)$.

As a consequence of the stabilization property, for subsequent invocations the effect of Algorithm 1 is to verify the goodness of the previously well-performed policies in $Smart$, then simulate the oldest policies in $Stale$, and then attempt to possibly select a good policy in $Poor$, by chance. Importantly yet counter-intuitively, a policy in $Poor$ can still deliver excellent performance in the future—it is sufficient that the workload changes in a way that plays to the strengths of this policy. In the scheduler, we set $\lambda = 0.6$, which means the top 60% of the simulated policies will be put into $Smart$ and the other 40% will be put into $Poor$. The setting is kept constant throughout the experiments and we will explore its impact in our future work.

5. EXPERIMENTAL SETUP

In this study, we use trace-based simulation to assess our portfolio scheduler and the proposed algorithm for time-constrained portfolio scheduling. We now present the simulator and the workload traces used for our experiments.

5.1 Simulator

To simulate the long-term execution of scientific workloads in the cloud, we extended our discrete event simulator DGSim [15] with entities such as a cloud-like resource manager and VM instances. The cloud-like resource manager implements Amazon EC2-style APIs for leasing and releasing VM instances, and implements the cost model of on-demand instances leased by Amazon EC2. The simulator used in this section should not be confused with the online simulator running as part of the portfolio scheduler. For the online simulator, it is tailored to be light-weight and can be easily integrated in other simulators such as CloudSim [4].

We simulate a cloud computing system where VM instances can be leased on-demand but the number of concurrently leased VMs is limited. This resource model is similar to the resource provisioning model enforced by our real-world DAS-4 system, which provides OpenNebula-based and Eucalyptus-based cloud interfaces to its users. Moreover, as declared by Amazon EC2, for the long-term usage of the cloud, reserved VM instances are much cheaper than their on-demand counterparts. In real virtualized environments, there is a delay for instance acquisition and booting, which is 90 to 120 seconds for Amazon EC2 instances and 4 minutes for our DAS-4 instances based on previous studies [13,23,41].

In all the experiments, we set the delay to 120 seconds and the maximum number of VMs that can be rented to 256. We further consider the functioning of a virtual cluster comprised of homogeneous VM instances. This model is consistent with the system configuration that the workload traces used in this work are produced. During the simulation, the jobs run exclusively on their VMs and cannot be preempted or migrated.

5.2 Workload Traces

We use in our simulations four traces from the Parallel Workloads Archive (PWA) [1]: KTH-SP2, SDSC-SP2, DAS2-fs0, and LPC-EGEE. We cleaned these traces by removing jobs with 0 runtime or processors, or with more than the maximum number of processors in the systems the traces were collected. We only use from these traces the jobs requesting up to 64 processors. Table 1 summarizes the characteristics of the cleaned traces—we use from each trace over 95% of the original jobs.

Figure 3 shows the arrival patterns of the submitted jobs for the four traces. KTH-SP2 exhibits stable arrivals, with few bursty moments; the SDSC-SP2 trace is similar. In contrast, the workloads of DAS2-fs0 and LPC-EGEE exhibit many bursty moments. For DAS2-fs0, the number of jobs during normal work hours is very small; for LPC-EGEE, the number of jobs during normal work hours is much larger than that in DAS2-fs0.

6. EXPERIMENTAL RESULTS

In this section, we study experimentally the use of portfolio scheduling for long-term execution of scientific workloads on IaaS cloud resources. We show the effects of workload patterns and of various parameter settings on the performance of our portfolio scheduler.

6.1 Effect of Portfolio Scheduling

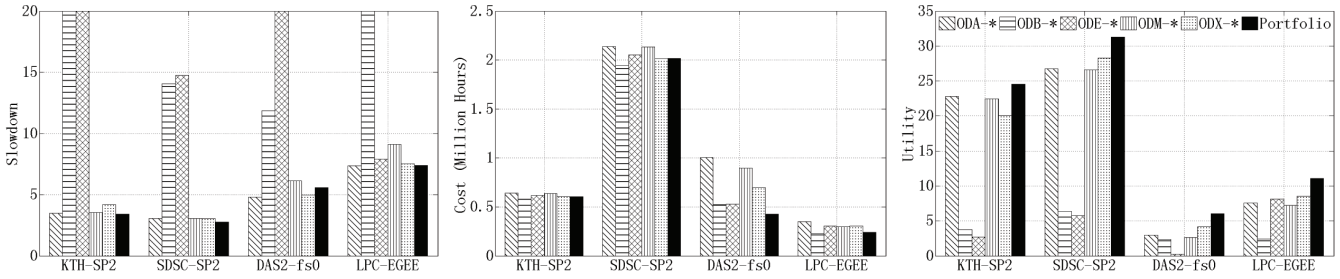
To show the effectiveness of our portfolio scheduler, we compare the performance of the portfolio scheduler with that of its constituent scheduling policies, taken independently. In this experiment, we use accurate runtime for the online simulator; we investigate the impact of inaccurate runtime prediction in Section 6.3. For clarity of depiction, we cluster the 60 scheduling policies by their provisioning policy and only plot in Figure 4 the best policy in each cluster. For example, in the cluster of provisioning policy ODA, there are 12 allocation policies (4 job selection policies times 3 VM selection policies); the scheduling policy with the highest utility is selected to represent the cluster and denoted as ODA-$*$. For all the 60 scheduling policies, we find that job selection policies such as UNICEF and LXF that favor short jobs have the best performance in all the clusters. Interestingly, instead of BestFit, FirstFit dominates the performance for VM selection.

Figure 4 compares the performance of our portfolio scheduler with that of the best constituent scheduling policies for our traces. As expected, our portfolio scheduler performs better than all individual scheduling policies, under all workload traces. In particular, it outperforms the best constituent policy under KTH-SP2, SDSC-SP2, DAS-fs0, and LPC-EGEE by 8%, 11%, 45%, and 30% respectively. Considering the characteristics of the traces, we can conclude that portfolio scheduling is effective especially for the burstier workloads, where the trace characteristics may change for each and in-between bursts. For the individual performance metrics, ODB and ODE have the largest job slowdown but relatively low charged cost. The reason is that the first two policies don't consider the wait time of the jobs but try to pack the jobs as tightly as possible. On the contrary, ODA, ODM, and ODX lease VMs either on-demand or with limited wait time, and have low job slowdown but relatively high charged cost.

To analyze the use of different policies by the portfolio scheduler, we plot the ratio of policy invocation in Figure 5. As shown by Figure 5(a), most of the scheduling policies have been invoked during the execution of the traces. For the invoked policies, the ratio is relatively even for KTH-SP2, SDSC-SP2 and DAS2-fs0. However, for LPC-EGEE several scheduling policies dominate the invocation process. We further coarsen the figure by merging the VM selection policies and job selection policies, in Figures 5(b) and 5(c), respectively. In Figure 5(c), ODB and ODX are the dominant provisioning policies for KTH-SP2 and SDSC-SP2—a result of the many long jobs, which can tolerate longer wait times. Since the majority of the jobs from DAS-fs0 and LPC-EGEE are very short, ODB and ODE become the dominant policies for these traces—it would be otherwise costly if VMs are rented on-demand or after a very short wait time—; however, ODX is also invoked frequently for the long jobs appearing in both DAS2-fs0 and LPC-EGEE.

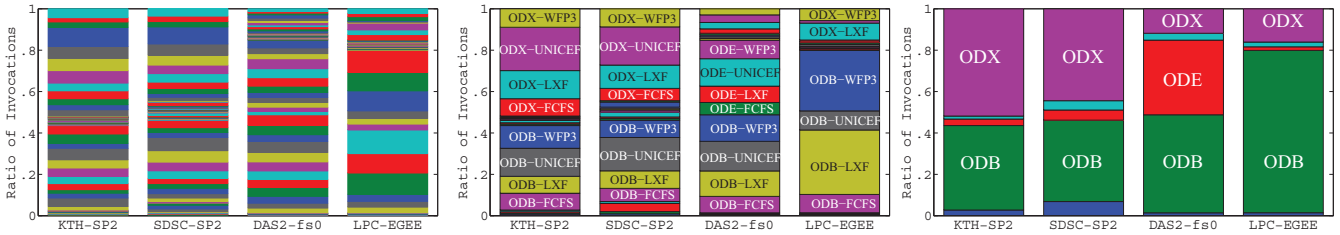
6.2 Effect of Utility Function

We now investigate the effect of the utility function parameters α and β on system performance and cost. First, we keep the task-urgency factor $\beta = 1$ and change the cost-efficiency factor α from 1 to 4. We also look at the extreme setting $\beta = 0$—ignoring the job slowdown. As depicted by Figure 6(b), we find that by emphasizing the cost-efficiency, the charged cost of all the traces decreases only slightly. Moreover, Figure 6(a) shows that the job slowdown of DAS2-fs0 and LPC-EGEE increases gradually. For $\beta = 0$, the job slowdown soars, yet the reduction of the charged cost is marginal.



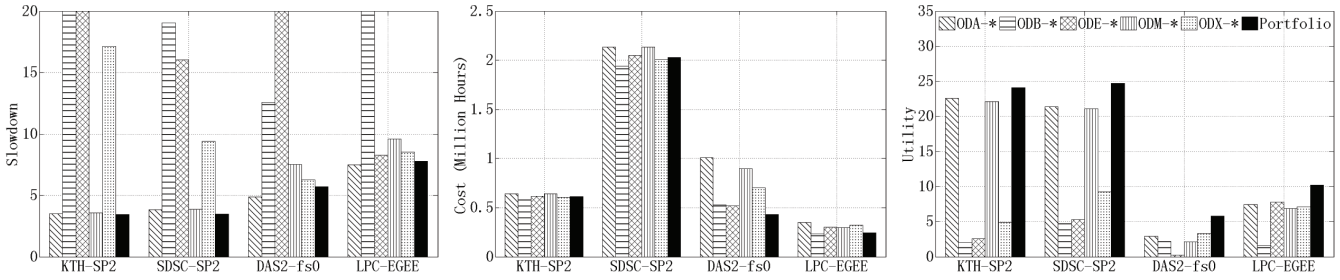
(a) Job slowdown. (b) Charged cost. (c) Utility function.

Figure 4: Performance of portfolio scheduling for accurate runtime. For KTH-SP2 and DAS2-fs0, the best allocation policy is UNICEF+FirstFit. For SDSC-SP2, the best allocation policy for ODA is UNICEF+BestFit; for others, it is UNICEF+FirstFit. For LPC-EGEE, the best allocation policy for ODM is LXF+WorstFit; for others, it is LXF+FirstFit.



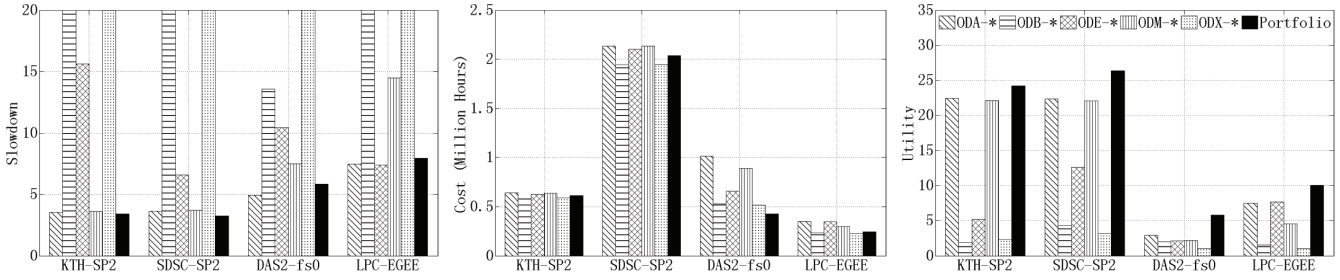
(a) 60 scheduling policies. (b) 5 provisioning × 4 job selection. (c) 5 provisioning policies.

Figure 5: The ratio of invocations of the scheduling policies. The order of the policies can be calculated by iterating the combination {ODA, ODB, ODE, ODM, ODX} × {FCFS, LXF, UNICEF, WFP3} × {BestFit, FirstFit, WorstFit}. We first show the result of all the 60 scheduling policies, then cluster the policies gradually for coarser granularity.



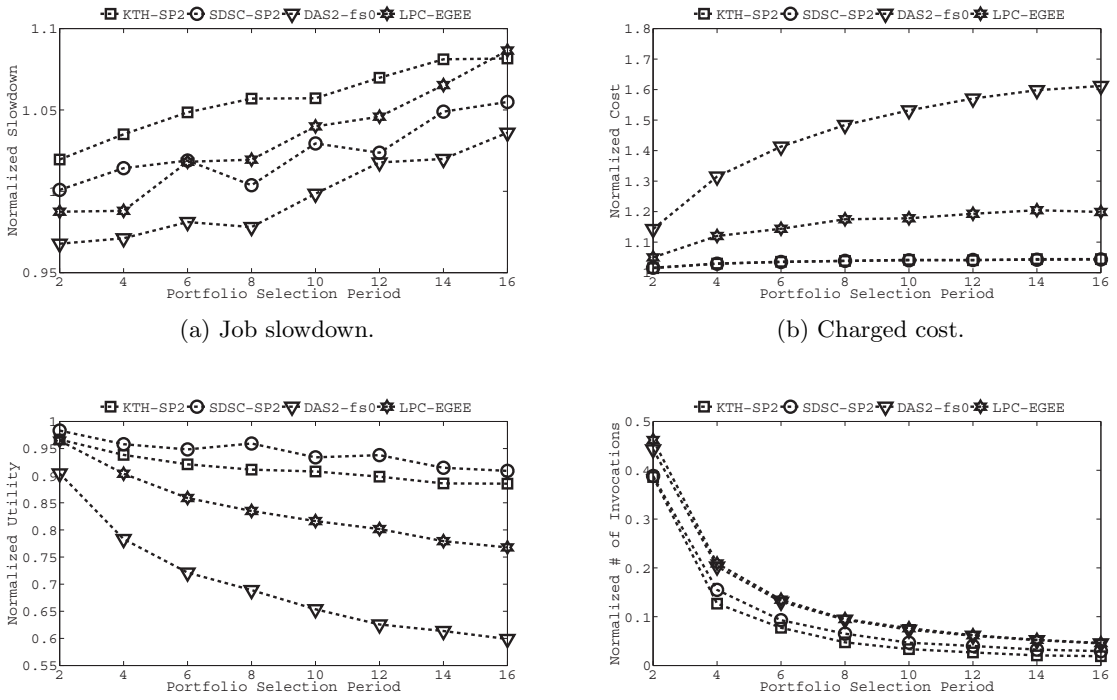
(a) Job slowdown. (b) Charged cost. (c) Utility function.

Figure 7: Performance of portfolio scheduling for predicted runtime. For SDSC-SP2 and DAS2-fs0, the best allocation policy is UNICEF+FirstFit. For KTH-SP2, the best allocation policy for ODA and ODM is UNICEF+BestFit. For LPC-EGEE, the best allocation policy for ODA and ODE is LXF+FirstFit; for others, it is LXF+WorstFit.



(a) Job slowdown. (b) Charged cost. (c) Utility function.

Figure 8: Performance of portfolio scheduling for user estimated runtime. For SDSC-SP2 and DAS2-fs0, the best allocation policy is UNICEF+FirstFit. For KTH-SP2, the best allocation policy for ODA is UNICEF+BestFit; for others, it is UNICEF+FirstFit. For LPC-EGEE, the best allocation policy for ODM and ODX is LXF+WorstFit; for others, it is LXF+FirstFit.



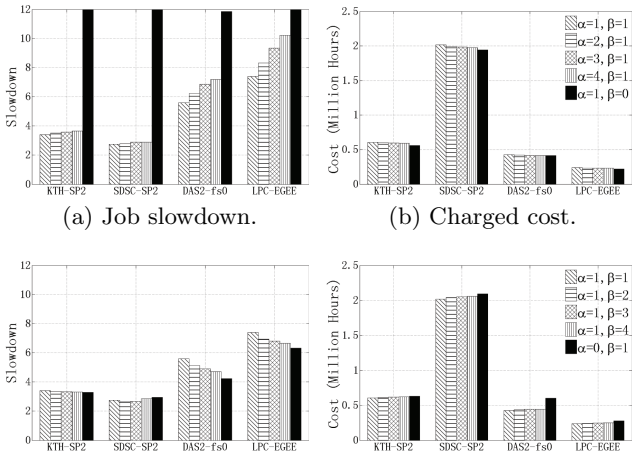
(a) Job slowdown.

(b) Charged cost.

(c) Utility function.

(d) Portfolio invocation.

Figure 9: The impact of portfolio selection period on performance. The selection period is a whole-number multiple of the scheduling period. The vertical axis does not start at 0.



(a) Job slowdown.

(b) Charged cost.

(c) Job slowdown.

(d) Charged cost.

Figure 6: The effect of the utility function. Top row: the cost-efficiency factor α varies. Bottom row: task-urgency factor β varies.

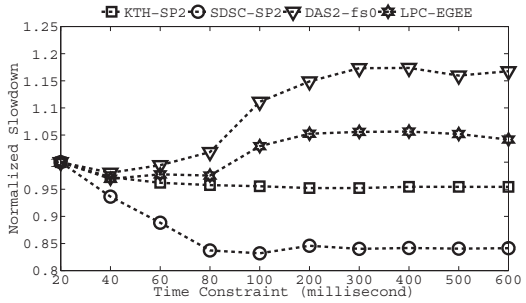
The second set of experiments varies the task-urgency factor, in the same way as cost-efficiency factor. From Figure 6(c), we see the job slowdown of DAS2-fs0 and LPC-EGEE declines considerably while the task-urgency factor increases. By ignoring the cost-efficiency ($\alpha = 0$), a minimum job slowdown is obtained for the two traces. However, as shown by Figure 6(d), DAS2-fs0 pays disproportionately for the reduction—about 40% more monetary cost in comparison with other cases. From both figures, we observe that the task-urgency has negligible impact on the performance of KTH-SP2 and SDSC-SP2.

Based on the results, we suggest that instead of putting effort to find sophisticated algorithms to reduce the cost, it is more worthwhile to find methods to improve the performance metrics that users are interested in, such as job slowdown and wait time. Such an observation is consistent with that of previous work [10]–the reason is the rather limited potential for cost improvement for long-term workloads in production systems.

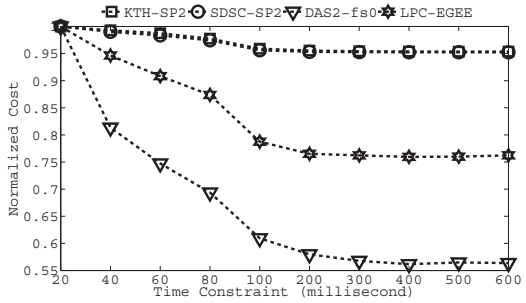
6.3 Impact of Prediction Inaccuracy

In the portfolio scheduler, some policies such as ODE and LXF need job runtime for calculation. Moreover, the online simulator also requires job runtime for simulation. In this subsection, we investigate the impact of inaccurate runtime prediction on the performance of portfolio scheduling. We use the runtime predicted by the method of Section 3.2 and that provided by the users for the evaluation. The results are shown in Figure 7 and Figure 8.

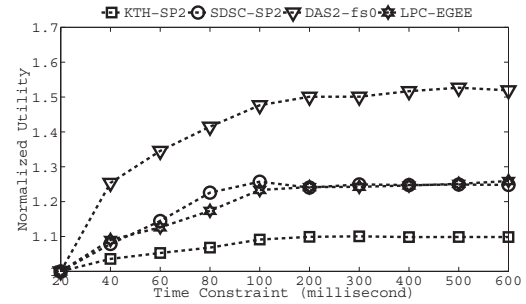
In comparison with Figure 4, inaccurate runtime prediction has an adverse effect on job slowdown of ODX especially while the runtime is provided by the users. Since user estimation is orders of magnitude larger than the actual runtime, VMs are overprovisioned by ODE. As a result, its job slowdown decreases significantly in Figure 8(a). Moreover, the charged cost of ODE increases while the large runtime is provided. The increment is significant in the DAS2-fs0 trace. Since jobs in ODX need to wait for a longer time, the corresponding cost is reduced considerably for DAS2-fs0 and LPC-EGEE while user estimated runtime is used.



(a) Job slowdown.



(b) Charged cost.



(c) Utility function.

Figure 10: The performance of portfolio scheduling under different time constraints. All the results are normalized by the result under the constraint of 20 milliseconds. The vertical axis does not start at 0.

It is interesting to see that our portfolio scheduler is not sensitive to the inaccurate runtime estimation. As shown by Figure 4, 7 and 8, only the job slowdown is slightly increased. In comparison with other policies, portfolio scheduling outperforms the best policies in KTH-SP2, SDSC-SP2, DAS-fs0, and LPC-EGEE by 6.87%, 15.6%, 77.32%, and 31.0% while using predicted runtime. While using user estimated runtime, the improvements are 7.72%, 18.04%, 101.07%, and 30.74% respectively. The performance improvement is not because scheduling policies can benefit from inaccurate runtime prediction, but policies using job runtime are adversely affected by inaccuracy. For example, the best policy for DAS2-fs0 is ODX-$*$ while actual runtime is used. However, while inaccurate runtime is used, ODA-$*$ becomes the best policy, leading to a relatively better performance of portfolio scheduling.

6.4 Impact of Portfolio Selection Period

The portfolio selection period is the interval between two consecutive selection processes. In the experiments conducted so far in Section 6, we have run the selection process every 20 seconds. In this section, we increase the selection period to measure its impact on the performance of our portfolio scheduler. Specifically, we run our portfolio scheduler under the four traces and set the selection period as a multiple of 2 to 16 times the 20-second period. We normalize the results with the results obtained for the scheduling period set to 20 seconds. We also plot the number of portfolio scheduling invocations during the execution of the whole traces, similarly normalized. Figure 9 depicts the results.

From Figure 9(a), we find that the portfolio selection period has an insignificant impact on job slowdown (less than 10% value change). The impact of the selection period on the charged cost is also negligible for KTH-SP2 and SDSC-SP2. However, for DAS2-fs0, the charged cost is greatly impacted by the selection period. When the selection period is set to 8 times the 20-second period, the charged cost increases by nearly 50%. For LPC-EGEE, the impact is moderate—the charged cost raises by up to about 15%.

The utility results, depicted in Figure 9(c), have an opposite trend in comparison with the charged cost. The number of invocations for portfolio selection decreases near-exponentially with the selection period. Combined with the performance data, we suggest a selection period of 8 for the stable KTH-SP2 and SDSC-SP2 traces. For LPC-EGEE, a selection period of 2 is suitable, since it reduces the number of portfolio selection by half but with a very small impact on the utility. However, for DAS2-fs0, to address the burstiness the scheduler needs to invoke the portfolio selection process at every scheduling period.

6.5 Impact of Simulation Time Constraint

Although the theoretical maximum computation complexity for the constituent scheduling policies is $\mathcal{O}(mn \log mn)$, where m is the number of queued jobs and n is the number of idle VMs, during the experiments we have observed that almost all the policies run without much overhead. In our experiments, often the values of m and n do not exceed 100. Therefore, to assess the performance impact of limiting the simulation time, we manually add a 10 milliseconds overhead for each of the scheduling policies. Figure 10 shows the performance results, normalized with the results obtained for a time constraint of 20 milliseconds.

As Figure 10(a) depicts, the job slowdown of the four traces shows different sensitivity to the time constraint. For KTH-SP2 and LPC-EGEE, the job slowdown has a marginal change (under 5%). For DAS2-fs0, the job slowdown declines slightly at the beginning and then increases to about 16% until the time constraint reaches about 300 milliseconds; afterwards, the slowdown stabilizes. SDSC-SP2 shows a trend opposite to DAS2-fs0's: the job slowdown decreases by up to about 15% for SDSC-SP2, until the time constraint reaches 80 milliseconds.

As shown by Figure 10(b), the charged cost exhibits a similar trend for all the traces, but with different amplitude. The charged cost of KTH-SP2 and SDSC-SP2 changes negligibly, by only about 5%. The charged cost of DAS2-fs0 and LPC-EGEE declines quickly until the time constraint reaches 100 milliseconds, and only slowly afterwards. The

charged cost is reduced by 20% and 40%, respectively.

The utility obtained for each trace exhibits similar trends when the simulation time constraint increases. The utility of all the traces increases until the time constraint reaches 200 milliseconds, then changes only slightly afterwards. Because a policy requires 10 milliseconds for simulation, the total number of simulated policies is 20. Therefore, for our portfolio scheduler, simulating a third of the total amount of policies (60) is sufficient. The reason can be found if we look back at Figure 5(b). As we pick 60% of the simulated policies as *Smart* policies, the number of policies in the *Smart* set would be 12. This covers almost all the dominant policies in Figure 5(b), which indicates the effectiveness of our time-constrained simulation algorithm.

7. RELATED WORK

In this section, we provide a review on research work related to three areas: computational portfolio design and portfolio-based algorithm selection, parallel job scheduling, and scientific workload scheduling in the cloud.

In 1976, Rice introduced the algorithm selection problem: with so many available algorithms, which one should be selected to solve the specific problem instance in order to optimize some performance objective [29]. Meanwhile, Rice presented an abstract model that can be used to explore the problem [29, 34]. However, the most widely-adopted solution to the problem follows a "winner-take-all" approach which selects the algorithm that has the best average performance for all performance instances [45]. In 1997, Huberman [12] introduced an economics approach based on portfolio theory [24] to the problem and proposed a general procedure for computational portfolio design. The idea is that by combining many algorithms into portfolios, a whole range of problem instances can be addressed. Our previous work [8] adapted the seminar idea and proposed a portfolio scheduler for scheduling scientific workloads in the data center; in parallel with our work but for a different setting, Shai et al. have looked at a type of portfolio scheduling without auto-tuning [30]. The focus of previous work is to answer the question how to use portfolio scheduling and if it works. In this article, we apply Rice's abstract model to portfolio scheduling and present a scheduler framework based the model for a more general and demanding application model. We explore the functionality of our scheduler and focus on the question: given dozens of policies, how to reduce the overhead of portfolio scheduling.

A large body of research work has been done on job scheduling in parallel computer systems. First-Come-First-Serve (FCFS) [18] is the earliest and simplest scheduling policy widely used in production batch systems. To reduce the fragmentation caused by head-of-line blocking in FCFS scheduling [3], EASY-Backfilling was introduced [20]. After that, many variants were designed to improve EASY-Backfilling [5, 19, 32, 37, 38, 44]. Among them, the adaptive scheduling policies are most relevant to our work [5, 19, 38]: they use different policies in different periods of time for scheduling; they also use online simulation to select the best policy for the next scheduling period. However, they are fundamentally different from our work. Firstly, they choose the policy that performs best for the previous workload, while our work chooses the best policy based on current workload. Secondly, instead of presenting a specifical scheduler, we proposed an abstract model and a framework for portfolio scheduling, which can be easily adapted to other scheduling areas. Last but not least, the major focus of our work is to explore portfolio scheduling itself and to find methods to reduce the overhead of scheduling in case the number of constituent policies are enormous.

The construction of our policy portfolio relies on previous studies related to utility-based job scheduling and scientific workload scheduling in the cloud. Similarly to our previous work [41], we divide the scheduling policy into three parts: resource provisioning, job selection, and VM selection. For the first part, we surveyed recent study on resource provisioning and allocation in the cloud, and selected the policies that both perform well and are suitable for parallel job scheduling for the construction of policy portfolio [8, 10, 25, 27, 41, 43]. The selected policies are also limited to identical type of on-demand VMs, and we refer to our prior work [31] for policies using multiple instance types. For job selection policies, we selected four utility functions from Tang's work [39] which covers most of the priority functions used in the literature. We don't consider backfilling in our current scheduling policies. We leave it for the future work and refer to [7] for its preliminary result on scheduling parallel jobs in the cloud. For VM selection policies, we selected three heuristics for the problem of bin packing [6]. We have briefly described the policies in Section 3.1 and more details can be found in the original work.

8. CONCLUSION AND FUTURE WORK

The elasticity of cloud computing and data center computing has shown great potential for scientific computing. In this article, we investigated the long-term execution of parallel scientific workloads on IaaS cloud resources. Instead of finding more sophisticated scheduling algorithms, we have proposed the use of portfolio scheduling, which combines many existing policies into a portfolio and selects online an appropriate policy to match specific workload and system conditions. We have first introduced an abstract model for the exploration of portfolio scheduling. Based on this model, we have designed a portfolio scheduling framework that includes several performance-affecting configuration parameters. Our framework combines over 60 policies—resource provisioning, job allocation, and VM allocation—into a single comprehensive portfolio. We have designed a versatile utility function and implemented an online simulator to select the right policy for the given workload. To address the critical issue of selecting an appropriate policy in time, we have proposed an algorithm for time-constrained policy selection, which aims at increasing the chance of selecting the best policy even under time limits that prevent the exhaustive evaluation of policies.

We have evaluated the performance of our portfolio scheduler through simulations, using four long-term real-world traces collected from parallel production environments. We have explored the impact on performance of different portfolio configurations, of inaccurate runtime prediction, and of different time constraints on policy selection. We found that (1) by exploiting the collective strengths of the constituent policies, our portfolio scheduler outperforms its best constituent policy; (2) although the charged cost for the traces used in our experiment is hardly reduced, as the system utilization is relatively high, it is still worthwhile to improve user-impacting performance such as the job slowdown; (3) while policies using job runtime

are badly affected by inaccurate information, our portfolio scheduler is much less sensitive; (4) for bursty workloads, the portfolio scheduler must make frequent decisions to achieve good performance; for stable workloads, long intervals between decisions suffice; (5) by clustering the policies into different categories, our simulation algorithm can reach good performance despite simulating only a few of the policies.

For the future, we want to find out whether and to what extend the reflection can help improve the quality of the selected policies. Secondly, we plan to develop an algorithm that can dynamically trigger the portfolio simulation process only when the workload pattern changes, thus reducing the number of invocations while preserving the performance. Thirdly, we intend to implement a real-world prototype of the scheduler and conduct realistic experiments to study performance of portfolio scheduling in practice. Finally, we are adapting portfolio scheduling for the execution of scientific workflows and expect that more application types can benefit from this new scheduling solution.

Acknowledgment

The authors would like to thank all the reviewers for their comments and positive feedbacks on our paper. This work is partially funded by the Dutch national research program COMMIT; and supported by the STW/NWO Veni grant 11881, the National Natural Science Foundation of China (Grant No. 60903042 and 61272483), and the R&D Special Fund for Public Welfare Industry (Meteorology) GYHY201306003.

9. REFERENCES

[1] Parallel workloads archive. http://www.cs.huji.ac.il/labs/parallel/workload/. 2013-02-17.

[2] O. Agmon Ben-Yehuda, A. Schuster, A. Sharov, M. Silberstein, and A. Iosup. Expert: Pareto-efficient task replication on grids and a cloud. In *IPDPS*, pages 167–178, 2012.

[3] A. AuYoung, A. Vahdat, and A. C. Snoeren. Evaluating the impact of inaccurate information in utility-based scheduling. In *SC*, 2009.

[4] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. D. Rose, and R. Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Softw., Pract. Exper.*, 41(1):23–50, 2011.

[5] S.-H. Chiang and S. Vasupongayya. Design and potential performance of goal-oriented job scheduling policies for parallel computer workloads. *IEEE Trans. Parallel Distrib. Syst.*, 19(12):1642–1656, 2008.

[6] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: a survey. In *Approximation algorithms for NP-hard problems*, pages 46–93. PWS Publishing Co., Boston, MA, USA, 1997.

[7] M. D. de Assunção, A. di Costanzo, and R. Buyya. Evaluating the cost-benefit of using cloud computing to extend the capacity of clusters. In *HPDC*, pages 141–150, 2009.

[8] K. Deng, R. Verboon, K. Ren, and A. Iosup. A periodic portfolio scheduler for scientific computing in the data center. In *JSSPP*, 2013.

[9] D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn. Parallel job scheduling - a status report. In *JSSPP*, pages 1–16, 2004.

[10] S. Genaud and J. Gossa. Cost-wait trade-offs in client-side resource provisioning with elastic clouds. In *IEEE CLOUD*, pages 1–8, 2011.

[11] T. J. Hacker and K. Mahadik. Flexible resource allocation for reliable virtual cluster computing systems. In *SC*, page 48, 2011.

[12] B. A. Huberman, R. M. Lukose, and T. Hogg. An economics approach to hard computational problems. *Science*, 275(5296):51–54, 1997.

[13] A. Iosup, S. Ostermann, N. Yigitbasi, R. Prodan, T. Fahringer, and D. H. J. Epema. Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE Trans. Parallel Distrib. Syst.*, 22(6):931–945, 2011.

[14] A. Iosup, O. O. Sonmez, S. Anoep, and D. H. J. Epema. The performance of bags-of-tasks in large-scale distributed systems. In *HPDC*, pages 97–108, 2008.

[15] A. Iosup, O. O. Sonmez, and D. H. J. Epema. Dgsim: Comparing grid resource management architectures through trace-based simulation. In *Euro-Par*, pages 13–25, 2008.

[16] K. Keahey, R. Figueiredo, J. Fortes, T. Freeman, and M. Tsugawa. Science clouds: Early experiences in cloud computing for scientific applications. *Cloud computing and applications*, 2008:16, 2008.

[17] K. Keahey and T. Freeman. Contextualization: Providing one-click virtual clusters. In *eScience*, pages 301–308, 2008.

[18] P. Krueger, T.-H. Lai, and V. A. Dixit-Radiya. Job scheduling is more important than processor allocation for hypercube computers. *IEEE Trans. Parallel Distrib. Syst.*, 5(5):488–497, 1994.

[19] B. Lawson and E. Smirni. Self-adaptive scheduler parameterization via online simulation. In *IPDPS*, 2005.

[20] D. A. Lifka. The anl/ibm sp scheduling system. In *JSSPP*, pages 295–303, 1995.

[21] M. Malawski, G. Juve, E. Deelman, and J. Nabrzyski. Cost- and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds. In *SC*, page 22, 2012.

[22] M. Mao and M. Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *SC*, page 49, 2011.

[23] M. Mao and M. Humphrey. A performance study on the vm startup time in the cloud. In *IEEE CLOUD*, pages 423–430, 2012.

[24] H. Markowitz. Portfolio selection*. *The journal of finance*, 7(1):77–91, 1952.

[25] P. Marshall, H. M. Tufo, and K. Keahey. Provisioning policies for elastic computing environments. In *IPDPS Workshops*, pages 1085–1094, 2012.

[26] A. M. Matsunaga and J. A. B. Fortes. On the use of machine learning to predict the time and resources consumed by applications. In *CCGRID*, pages 495–504, 2010.

[27] E. Michon, J. Gossa, and S. Genaud. Free elasticity

and free cpu power for scientific workloads on iaas clouds. In *ICPADS*, pages 85–92, 2012.

[28] A.-M. Oprescu, T. Kielmann, and H. Leahu. Stochastic tail-phase optimization for bag-of-tasks execution in clouds. In *UCC*, pages 204–208, 2012.

[29] J. R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.

[30] O. Shai, E. Shmueli, and D. G. Feitelson. Heuristics for resource matching in intel's compute farm. In *JSSPP*, 2013.

[31] S. Shen, K. Deng, A. Iosup, and D. H. J. Epema. Scheduling jobs in the cloud using on-demand and reserved instances. In *Euro-Par*, pages 242–254, 2013.

[32] E. Shmueli and D. G. Feitelson. Backfilling with lookahead to optimize the packing of parallel jobs. *J. Parallel Distrib. Comput.*, 65(9):1090–1107, 2005.

[33] W. Smith. Prediction services for distributed computing. In *IPDPS*, pages 1–10, 2007.

[34] K. Smith-Miles. Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Comput. Surv.*, 41(1), 2008.

[35] O. O. Sonmez, N. Yigitbasi, S. Abrishami, A. Iosup, and D. H. J. Epema. Performance analysis of dynamic workflow scheduling in multicluster grids. In *HPDC*, pages 49–60, 2010.

[36] O. O. Sonmez, N. Yigitbasi, A. Iosup, and D. H. J. Epema. Trace-based evaluation of job runtime and queue wait time predictions in grids. In *HPDC*, pages 111–120, 2009.

[37] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan. Selective reservation strategies for backfill job scheduling. In *JSSPP*, pages 55–71, 2002.

[38] D. Talby and D. G. Feitelson. Improving and stabilizing parallel computer performance using adaptive backfilling. In *IPDPS*, 2005.

[39] W. Tang, Z. Lan, N. Desai, and D. Buettner. Fault-aware, utility-based job scheduling on blue, gene/p systems. In *CLUSTER*, pages 1–10, 2009.

[40] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Trans. Parallel Distrib. Syst.*, 18(6):789–803, 2007.

[41] D. Villegas, A. Antoniou, S. M. Sadjadi, and A. Iosup. An analysis of provisioning and allocation policies for infrastructure-as-a-service clouds. In *CCGRID*, pages 612–619, 2012.

[42] G. von Laszewski, J. Diaz, F. Wang, and G. Fox. Comparison of multiple cloud frameworks. In *IEEE CLOUD*, pages 734–741, 2012.

[43] L. Wang, J. Zhan, W. Shi, and Y. Liang. In cloud, can scientific communities benefit from the economies of scale? *IEEE Trans. Parallel Distrib. Syst.*, 23(2):296–303, 2012.

[44] A. M. Weil and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *IEEE Trans. Parallel Distrib. Syst.*, 12(6):529–543, 2001.

[45] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Satzilla: Portfolio-based algorithm selection for sat. *J. Artif. Intell. Res. (JAIR)*, 32:565–606, 2008.